

Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Fall & Winter 2025
Lecture 3

Priority Queues

- Review of binary heaps
- Leftist heaps
- Skew heaps
- Binomial queues
- Take-home messages

Priority Queues

- Review of binary heaps
- Leftist heaps
- Skew heaps
- Binomial queues
- Take-home messages

Priority queue data type

A min-oriented priority queue supports the following core operations:

- MAKE-HEAP(): create an empty heap.
- INSERT(H, x): insert an element x into the heap.
- EXTRACT-MIN(H): remove and return an element with the smallest key.
- DECREASE-KEY(H, x, k): decrease the key of element x to k .

The following operations are also useful:

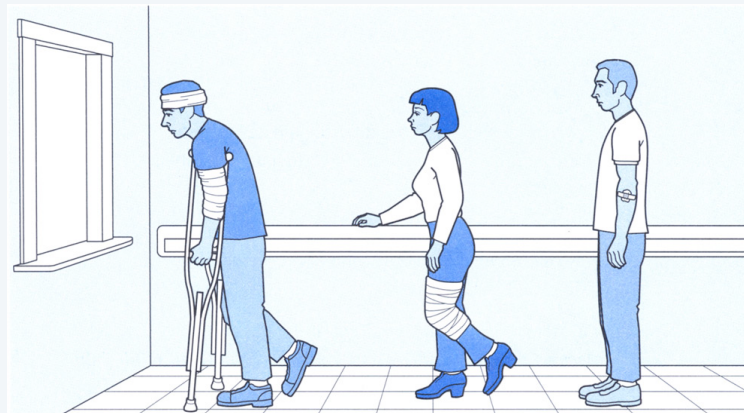
- IS-EMPTY(H): is the heap empty?
- FIND-MIN(H): return an element with smallest key.
- DELETE(H, x): delete element x from the heap.
- MELD(H_1, H_2): replace heaps H_1 and H_2 with their union.

Note. Each element contains a key (duplicate keys are permitted) from a totally-ordered universe.

Priority queue applications

Applications.

- A* search.
- Heapsort.
- Online median.
- Huffman encoding.
- Prim's MST algorithm.
- Discrete event-driven simulation.
- Network bandwidth management.
- Dijkstra's shortest-paths algorithm.
- ...

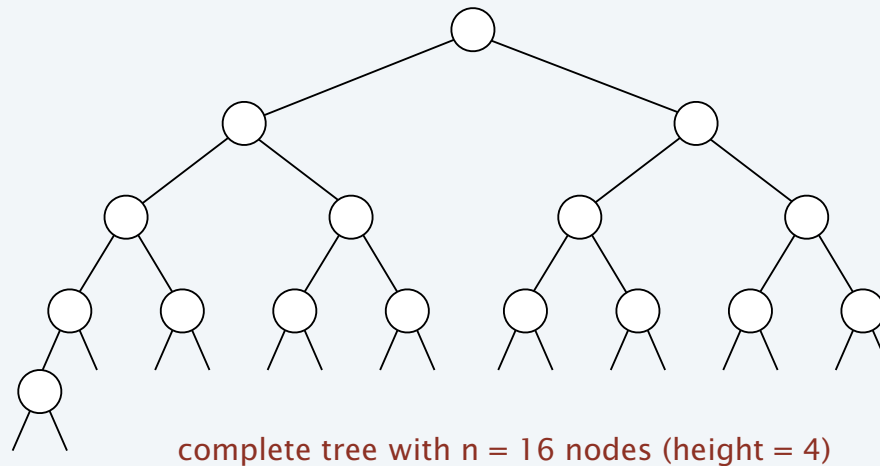


<http://younginc.site11.com/source/5895/fos0092.html>

Complete binary tree

Binary tree. Empty or node with links to two disjoint binary trees.

Complete tree. Perfectly balanced, except for bottom level.



Property. Height of complete binary tree with n nodes is $\lfloor \log_2 n \rfloor$.

Pf. Height increases (by 1) only when n is a power of 2. ■

A complete binary tree in nature



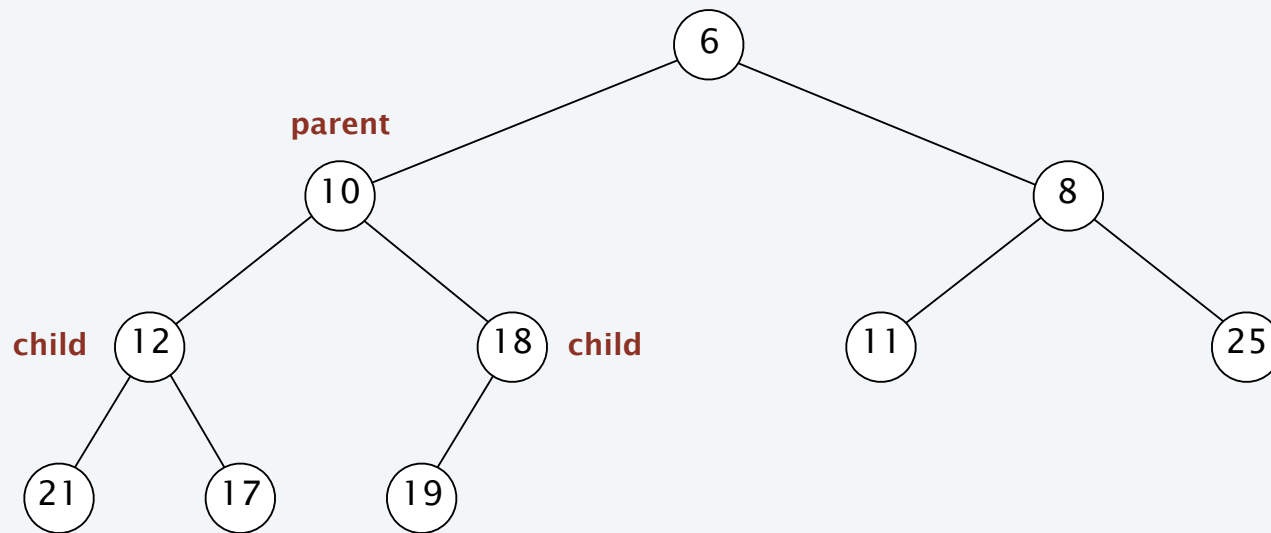
Hyphaene Compressa - Doum Palm

© Shlomit Pinter

Binary heap

Binary heap. Heap-ordered complete binary tree.

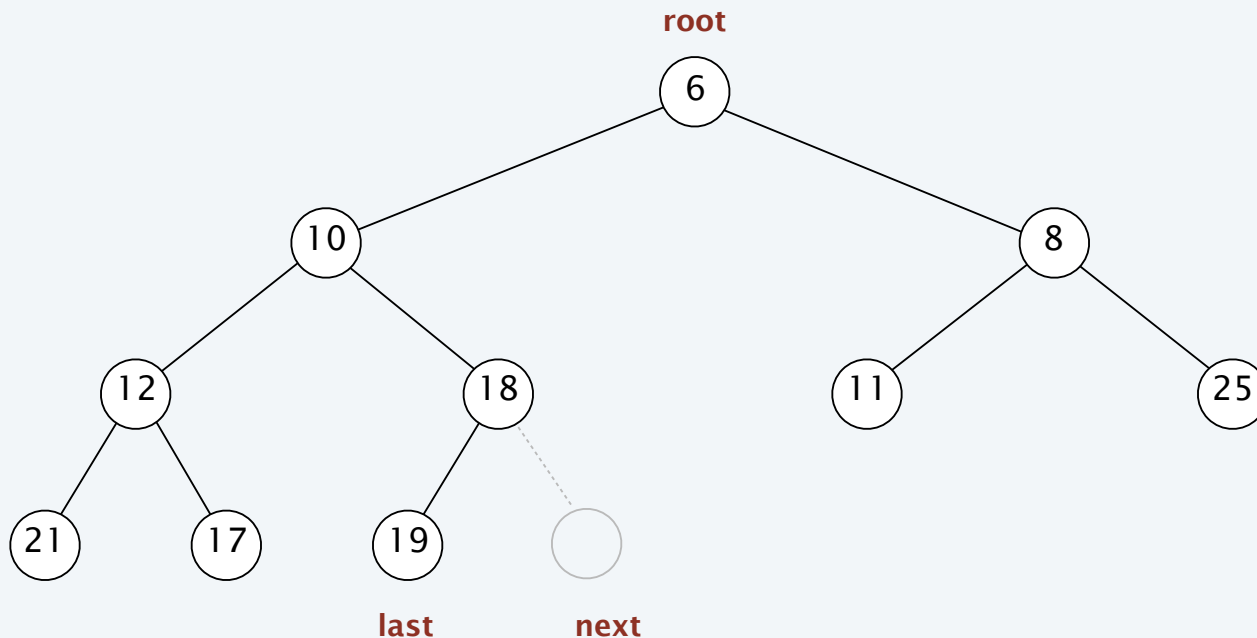
Heap-ordered tree. For each child, the key in child \geq key in parent.



Explicit binary heap

Pointer representation. Each node has a pointer to parent and two children.

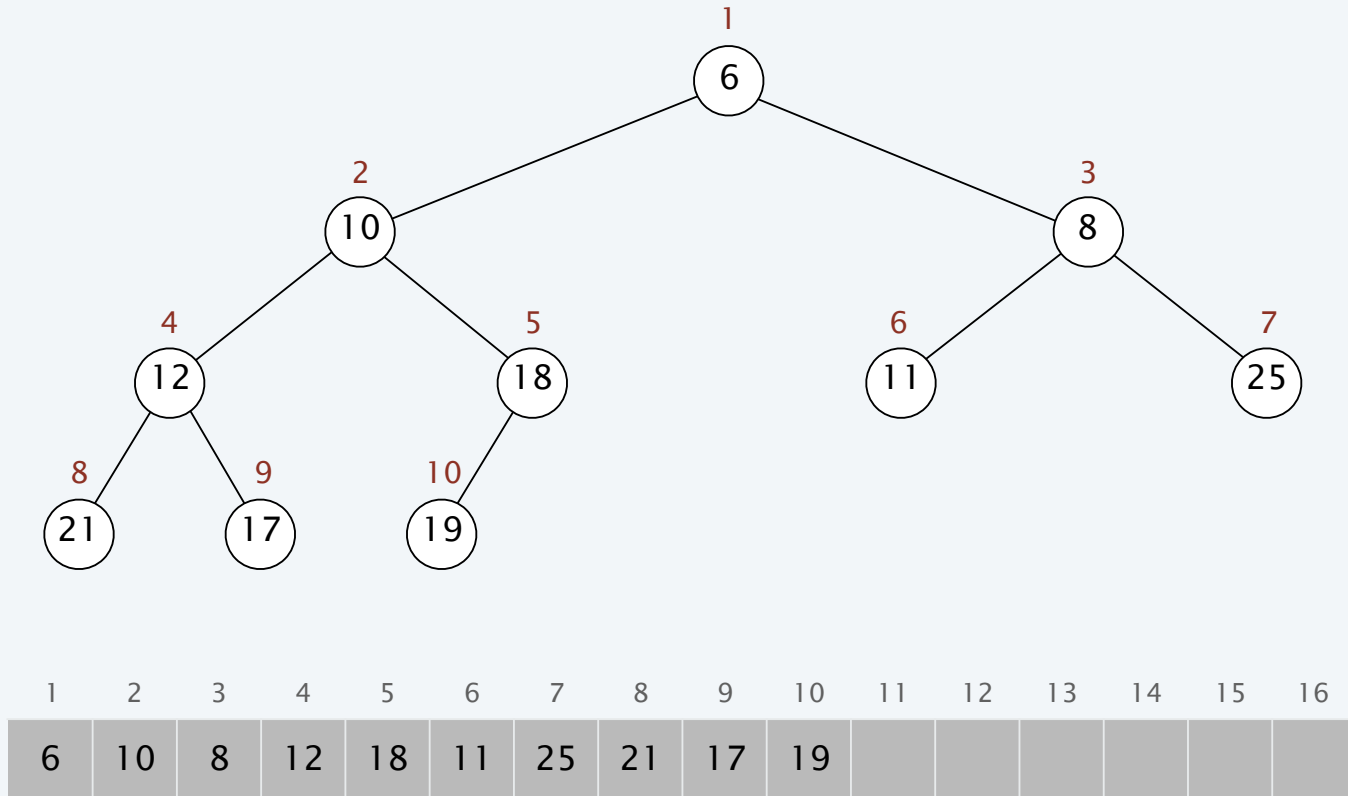
- Maintain number of elements n .
- Maintain pointer to root node.
- Can find pointer to last node or next node in $O(\log n)$ time.



Implicit binary heap

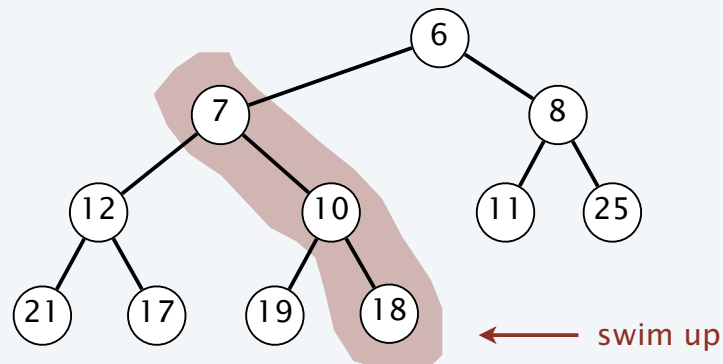
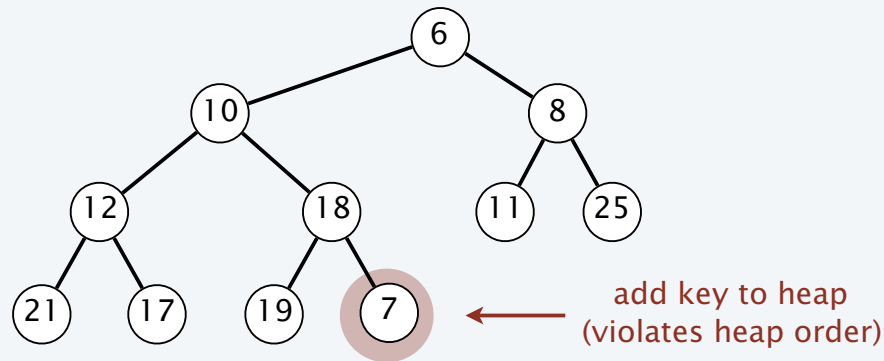
Array representation. Indices start at 1.

- Take nodes in **level** order.
- Parent of node at k is at $\lfloor k/2 \rfloor$.
- Children of node at k are at $2k$ and $2k+1$.



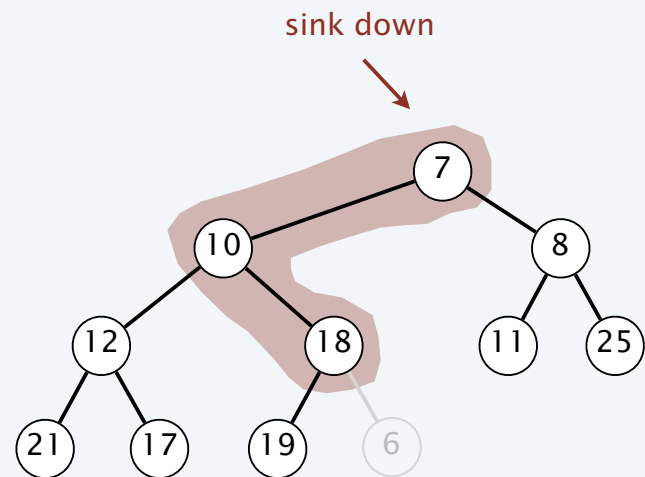
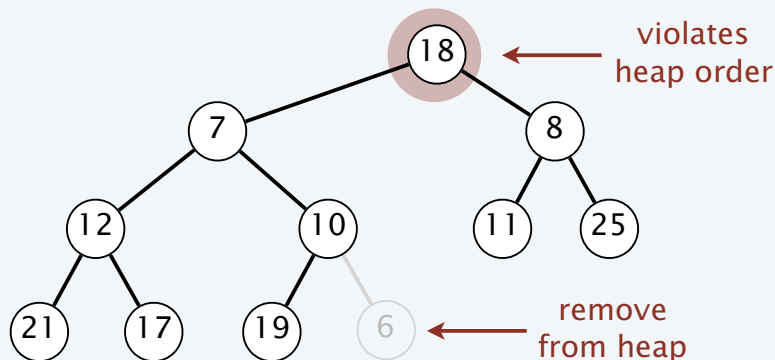
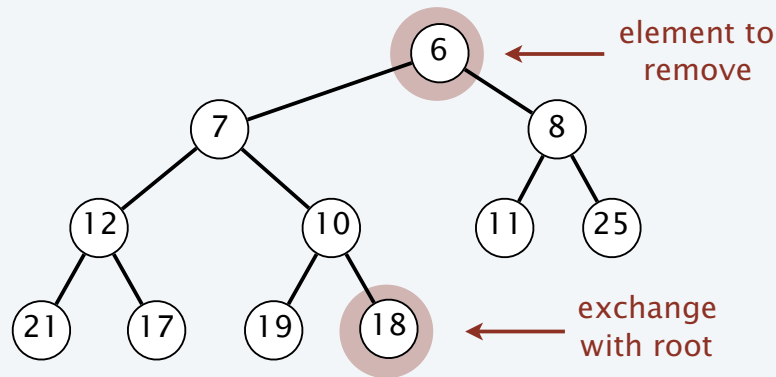
Binary heap: insert

Insert. Add element in new node at end; repeatedly exchange new element with element in its parent until heap order is restored.



Binary heap: extract the minimum

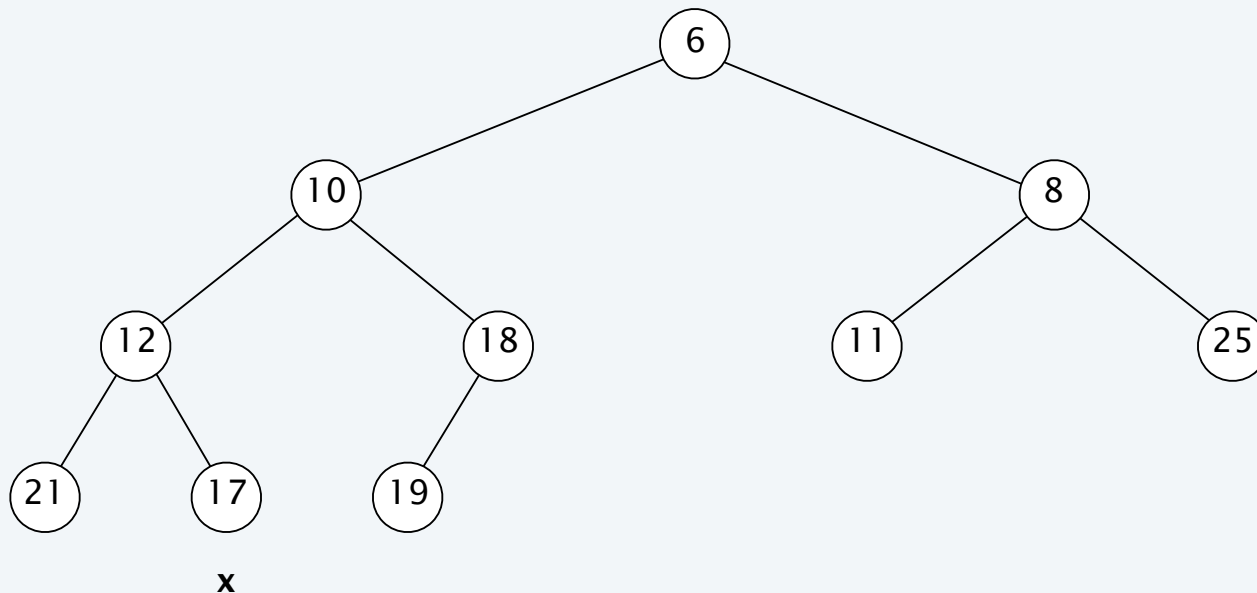
Extract min. Exchange element in root node with last node; repeatedly exchange element in root with its smaller child until heap order is restored.



Binary heap: decrease key

Decrease key. Given a **handle** to node, repeatedly exchange element with its parent until heap order is restored.

decrease key of node x to 11



Binary heap: analysis

Theorem. In an **implicit** binary heap, any sequence of m INSERT, EXTRACT-MIN, and DECREASE-KEY operations with n INSERT operations takes $O(m \log n)$ time.

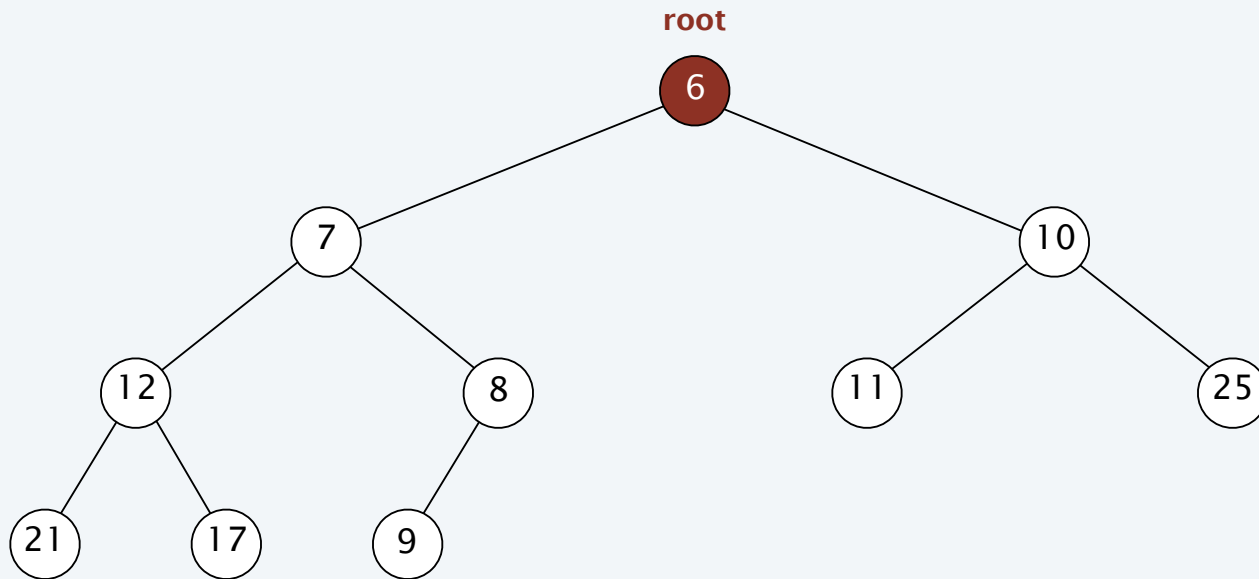
Pf.

- Each heap op touches nodes only on a path from the root to a leaf; the height of the tree is at most $\log_2 n$.
- The total cost of expanding and contracting the arrays is $O(n)$. ▀

Theorem. In an **explicit** binary heap with n nodes, the operations INSERT, DECREASE-KEY, and EXTRACT-MIN take $O(\log n)$ time in the worst case.

Binary heap: find-min

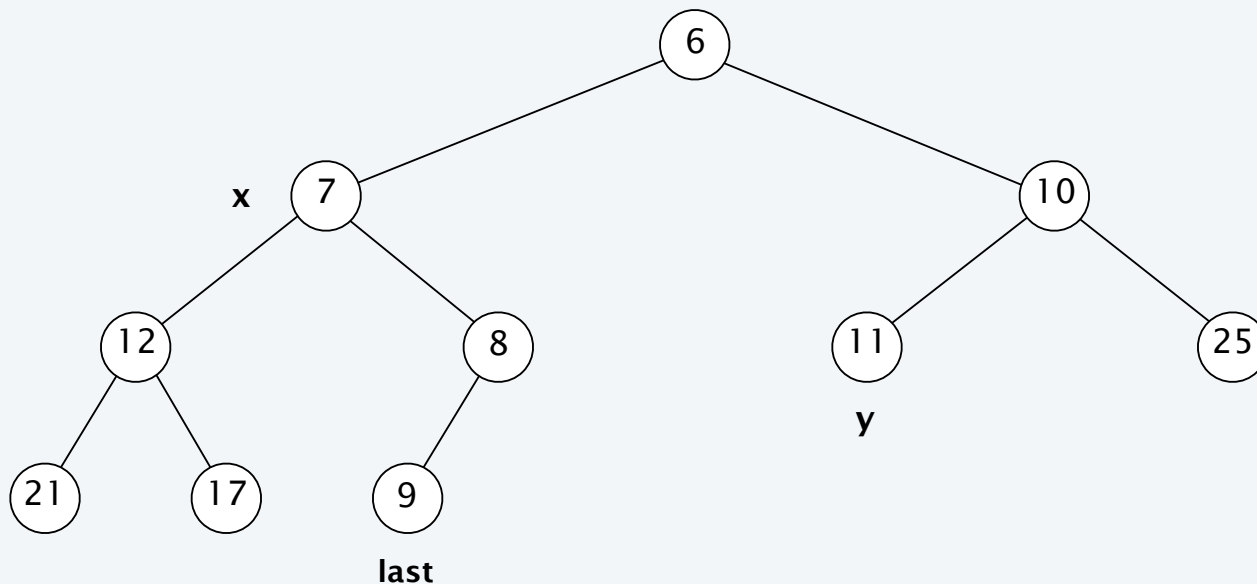
Find the minimum. Return element in the root node.



Binary heap: delete

Delete. Given a **handle** to a node, exchange element in node with last node; either swim down or sink up the node until heap order is restored.

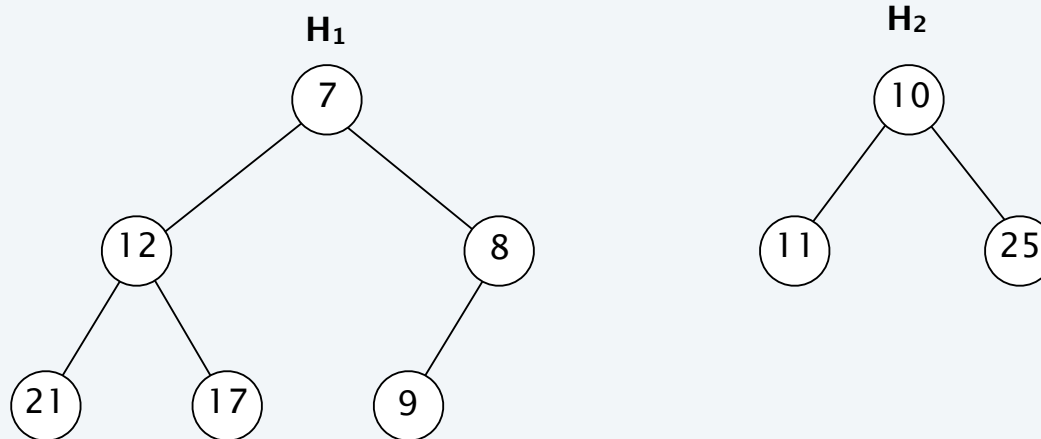
delete node x or y



Binary heap: meld

Meld. Given two binary heaps H_1 and H_2 , merge into a single binary heap.

Observation. No easy solution: $\Omega(n)$ time apparently required.

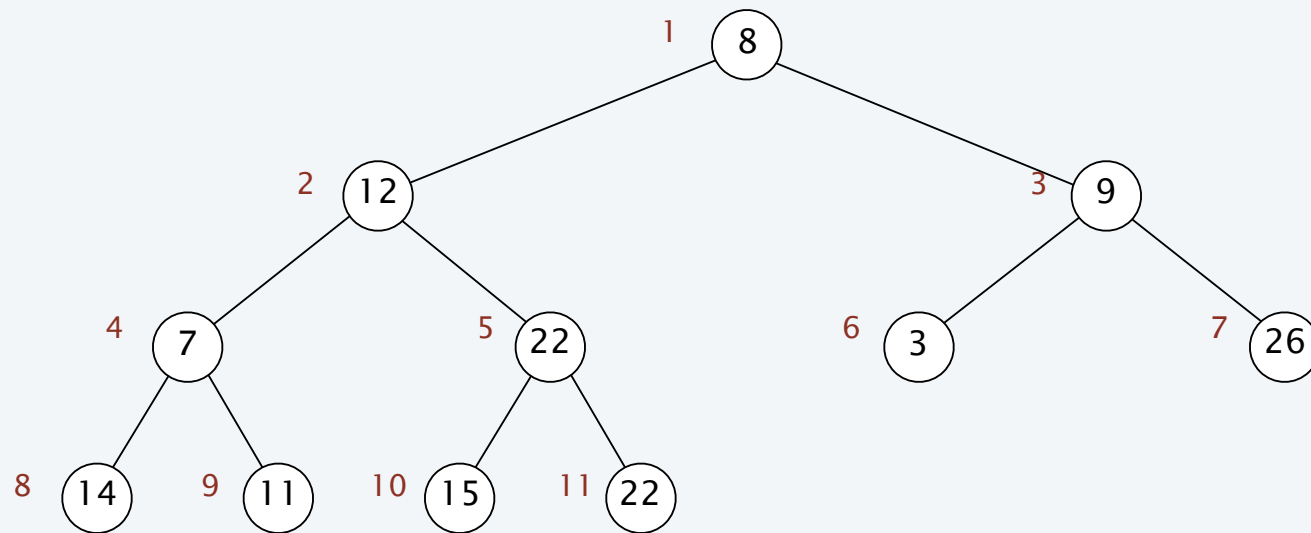


Binary heap: heapify

Heapify. Given n elements, construct a binary heap containing them.

Observation. Can do in $O(n \log n)$ time by inserting each element.

Bottom-up method. For $i = n$ to 1, repeatedly exchange the element in node i with its smaller child until subtree rooted at i is heap-ordered.



8	12	9	7	22	3	26	14	11	15	22
1	2	3	4	5	6	7	8	9	10	11



Binary heap: heapify

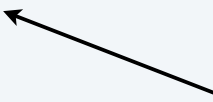
Theorem. Given n elements, can construct a binary heap containing those n elements in $O(n)$ time.

Pf.

- There are at most $\lceil n / 2^{h+1} \rceil$ nodes of height h .
- The amount of work to sink a node is proportional to its height h .
- Thus, the total work is bounded by:

$$\begin{aligned} \sum_{h=0}^{\lfloor \log_2 n \rfloor} \lceil n / 2^{h+1} \rceil h &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} n h / 2^h \\ &\leq 2n \quad \blacksquare \end{aligned}$$

$\sum_{i=1}^k \frac{i}{2^i} = 2 - \frac{k}{2^k} - \frac{1}{2^{k-1}} \leq 2$



Corollary. Given two binary heaps H_1 and H_2 containing n elements in total, can implement MELD in $O(n)$ time.

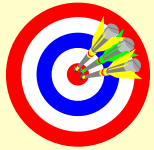
Priority queues performance cost summary

operation	linked list	binary heap
MAKE-HEAP	$O(1)$	$O(1)$
ISEMPTY	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$
EXTRACT-MIN	$O(n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$
DELETE	$O(1)$	$O(\log n)$
MELD	$O(1)$	$O(n)$
FIND-MIN	$O(n)$	$O(1)$

Priority Queues

- Review of binary heaps
- **Leftist heaps**
- Skew heaps
- Binomial queues
- Take-home messages

Leftist Heaps



Target: Speed up merging in $O(N)$.

👁️ **Heap: Structure Property + Order Property**

👎 Have to copy one array into another ➡ $\Theta(N)$

👉 Use **pointers** 👎 Slow down all the operations

Leftist Heap:

Order Property – the same

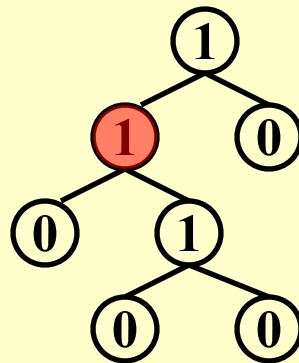
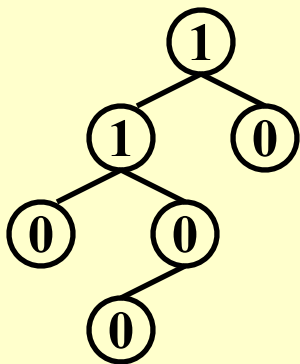
Structure Property – binary tree, but *unbalanced*

[Definition] The **null path length**, $Npl(X)$, of any node X is the length of the shortest path from X to a node without two children. Define $Npl(NULL) = -1$.

Note:

$$Npl(X) = \min \{ Npl(C) + 1 \text{ for all } C \text{ as children of } X \}$$

[Definition] The **leftist heap property** is that for every node X in the heap, the null path length of the **left** child is **at least as large as** that of the **right** child.



The tree is biased to get deep toward the *left*.

[Theorem] A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

Proof: By induction on p. 162.

Note: The leftist tree of N nodes has a right path containing at most $\lfloor \log(N+1) \rfloor$ nodes.



We can perform all the work on the *right* path, which is guaranteed to be *short*.

Trouble makers: Insert and Merge

Note: Insertion is merely a special case of merging.

Leftist trees have a short path

Thm. If rightmost path of leftist tree has r nodes, then whole tree has at least $2^r - 1$ nodes.

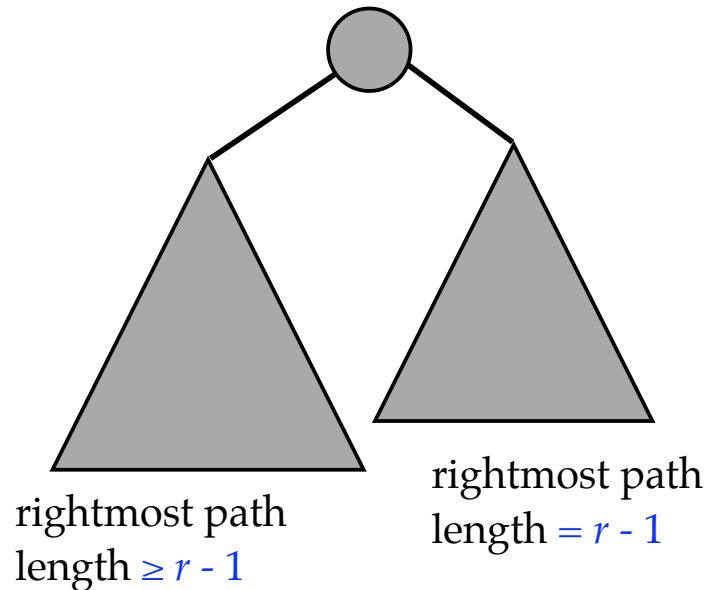
Proof.

Base Case: When $r = 1$, $2^1 - 1 = 1$ & tree has ≥ 1 node.

Induction hypothesis: Assume
 $N(i) \geq 2^i - 1$ for $i < r$.

Induction step: Left and right subtrees of the root have at least $2^{r-1} - 1$, nodes.

Thus, at least $2(2^{r-1} - 1) + 1 = 2^r - 1$ nodes in original tree. \square

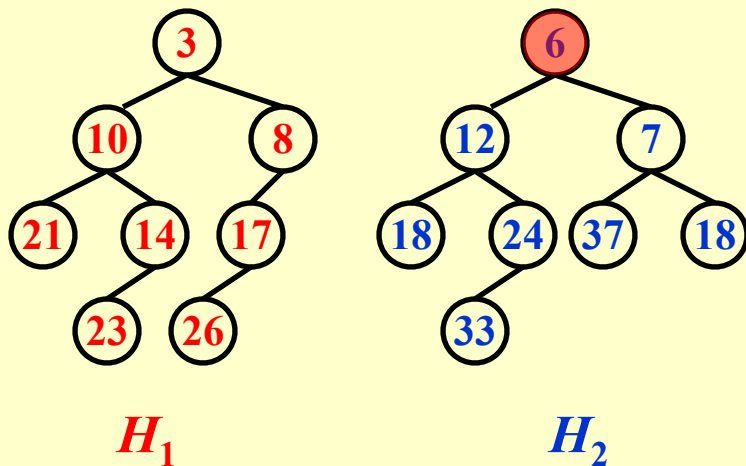


Therefore $n \geq 2^r - 1$, so r is $O(\log n)$

Declaration:

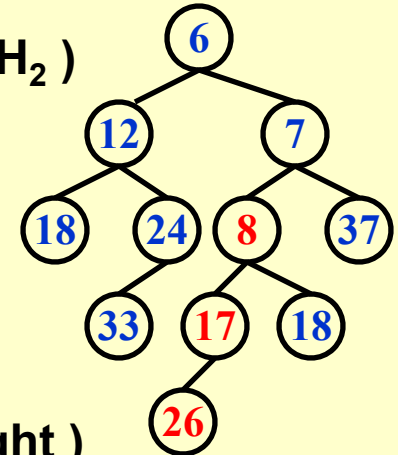
```
struct TreeNode
{
    ElementType    Element;
    PriorityQueue  Left;
    PriorityQueue  Right;
    int           Npl;
};
```

Merge (recursive version):



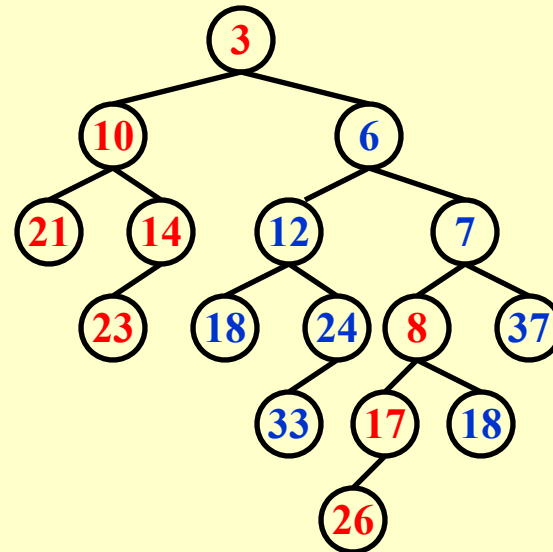
Step 1:

Merge($H_1 \rightarrow \text{Right}$, H_2)



Step 2:

Attach(H_2 , $H_1 \rightarrow \text{Right}$)



Step 3:

Swap($H_1 \rightarrow \text{Right}$, $H_1 \rightarrow \text{Left}$)
if necessary

```

PriorityQueue Merge ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1 == NULL ) return H2;
    if ( H2 == NULL ) return H1;
    if ( H1->Element < H2->Element ) return Merge1( H1, H2 );
    else return Merge1( H2, H1 );
}

```

```

static PriorityQueue
Merge1( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1->Left == NULL )    /* single node */
        H1->Left = H2;        /* H1->Right is already NULL
                               and H1->Npl is already 0 */

    else {
        H1->Right = Merge( H1->Right, H2 );    /* Step 1 & 2 */
        if ( H1->Left->Npl < H1->Right->Npl )
            SwapChildren( H1 );                /* Step 3 */
        H1->Npl = H1->Right->Npl + 1;

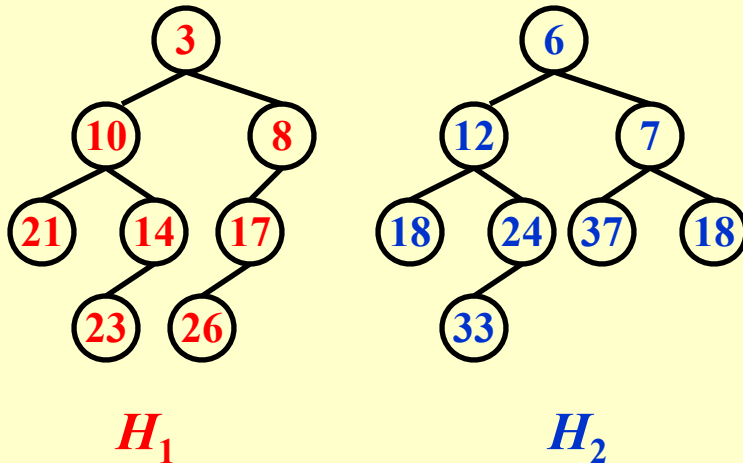
    } /* end else */
    return H1;
}

```

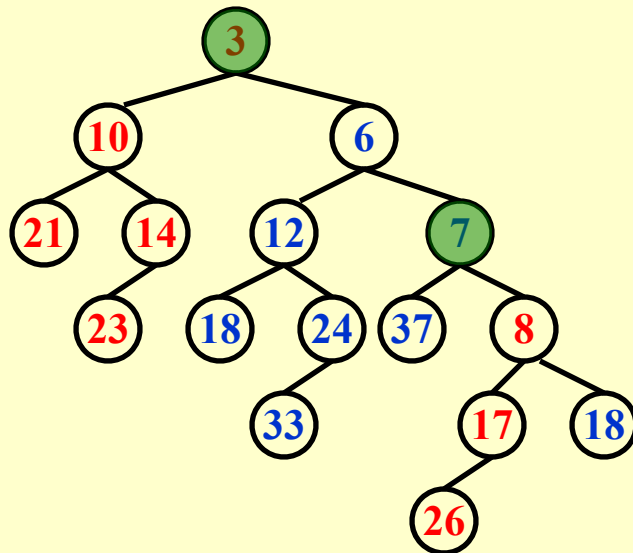
$$T_p = O(\log N)$$

What if *Npl* is NOT updated?

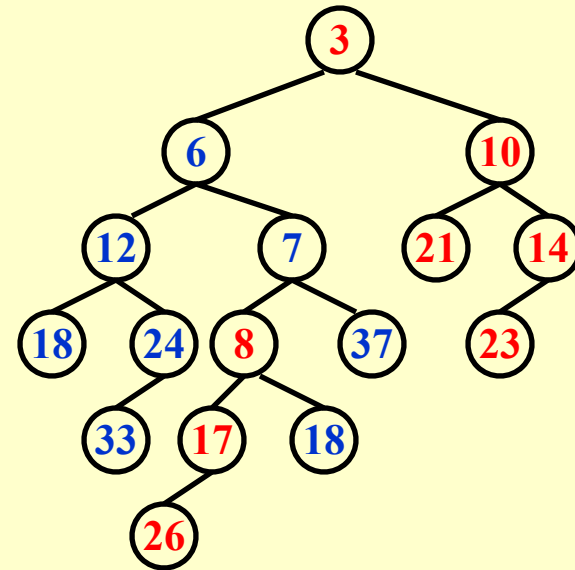
👉 Merge (iterative version):



Step 1: Sort the right paths without changing their left children



Step 2: Swap children if necessary



👉 DeleteMin:

Step 1: Delete the root

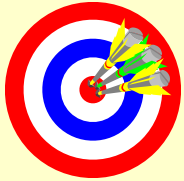
Step 2: Merge the two subtrees

$$T_p = O(\log N)$$

Priority Queues

- Review of Binary Heaps
- Leftist Heaps
- **Skew Heaps**
- Binomial queues
- Take-home messages

Skew Heaps -- a simple version of the leftist heaps

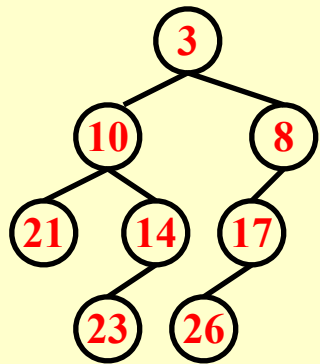


Target: Any M consecutive operations take at most $O(M \log N)$ time.

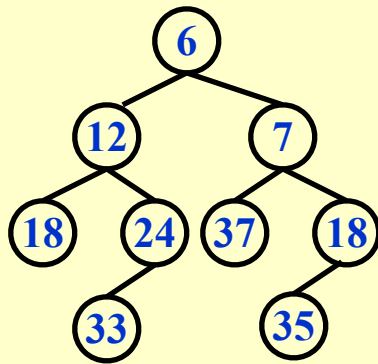
👉 **Merge:** **Always** swap the left and right children except that the **largest** of all the nodes on the right paths does not have its children swapped. **No Npl.**

even when one tree is finished, subtree from another tree continues merging.

(see the insert example in the next slide)

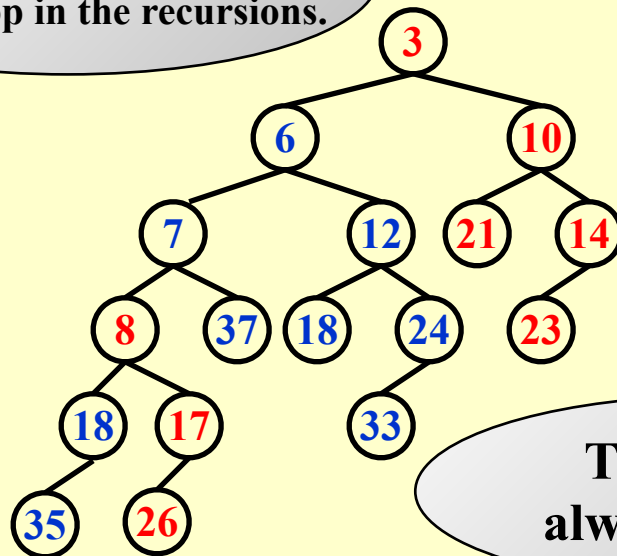


H_1



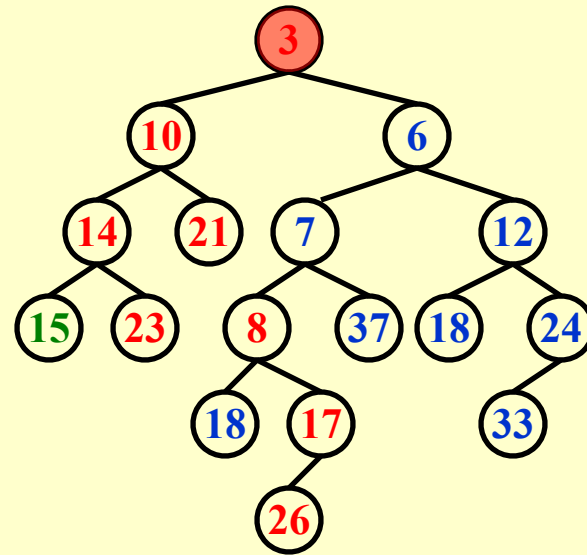
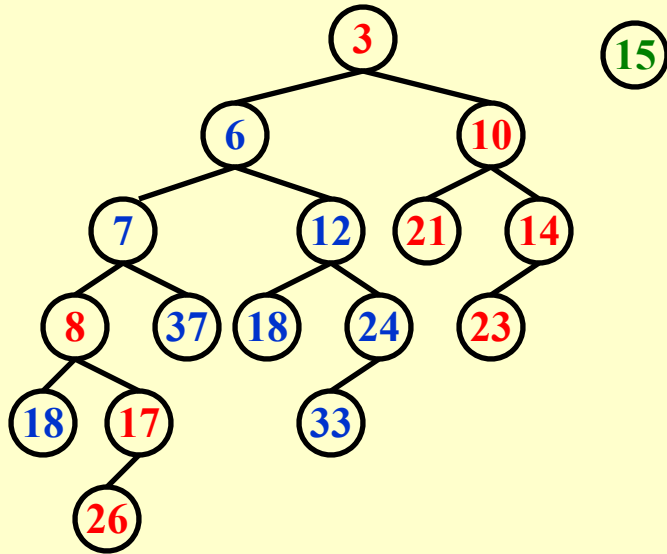
H_2

Not really a special case, but a natural stop in the recursions.

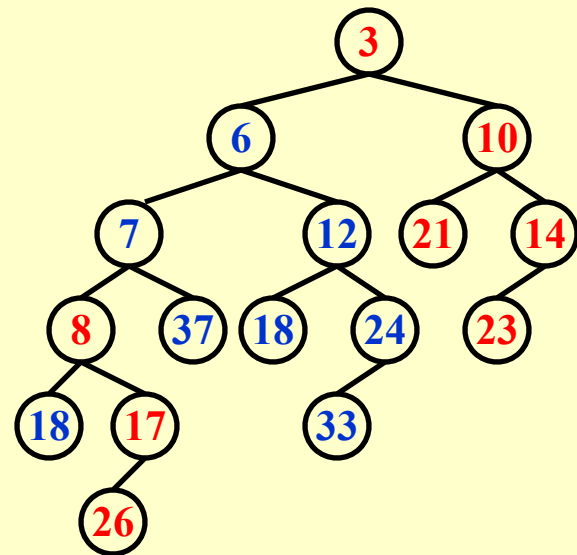
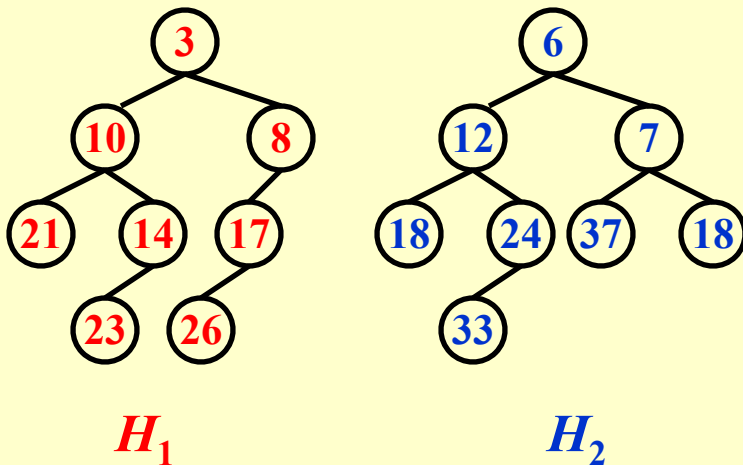


This is NOT always the case.

【 Example 】 Insert 15



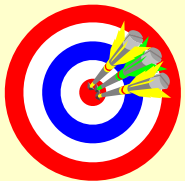
👉 Merge (iterative version):



Note:

☞ Skew heaps have the advantage that **no extra space** is required to maintain path lengths and **no tests** are required to determine when to swap children.

☞ It is an open problem to determine precisely the **expected right path length** of both leftist and skew heaps.



Target : Any M consecutive operations take at most $O(M \log N)$ time.

How to prove this?

Amortized Analysis for Skew Heaps

Insert & Delete are just **Merge**

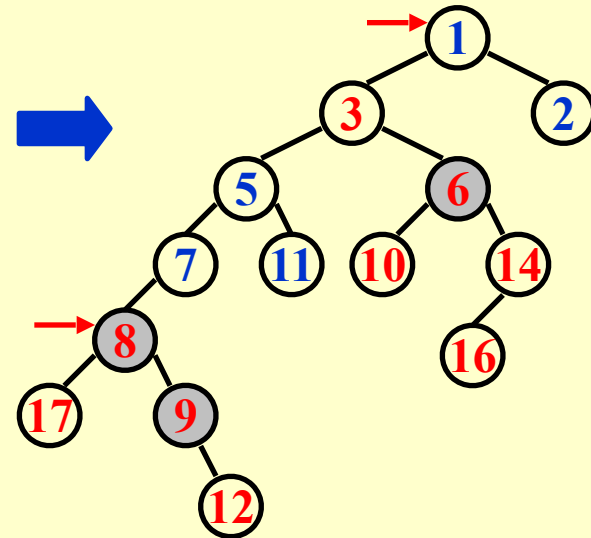
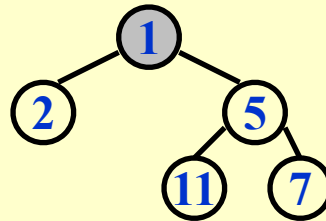
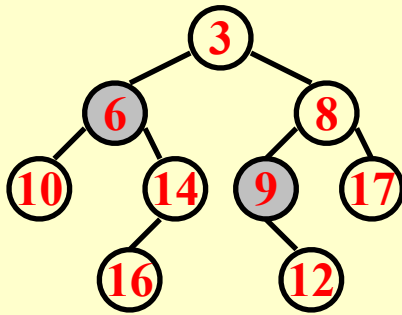
$$T_{\text{amortized}} = O(\log N) ?$$

D_i = the root of the resulting tree

$\Phi(D_i)$ = number of *heavy* nodes

#right node does not necessarily decrease after merge. Usually even increases.

【Definition】 A node p is *heavy* if the number of descendants of p 's right subtree is at least half of the number of descendants of p , and *light* otherwise. Note that the number of descendants of a node includes the node itself.



The only nodes whose heavy/light status can change are nodes that are initially on the right path.

$$H_i : l_i + h_i \quad (i = 1, 2)$$



$$T_{worst} = l_1 + h_1 + l_2 + h_2$$

Along the right path

Before merge: $\Phi_i = h_1 + h_2 + h_{other}$

$$T_{amortized} = T_{worst} + \Phi_{i+1} - \Phi_i$$

After merge: $\Phi_{i+1} \leq l_1 + l_2 + h_{other}$

$$\leq 2(l_1 + l_2)$$

$$l = O(\log N)$$



$$T_{amortized} = O(\log N)$$

Priority Queues

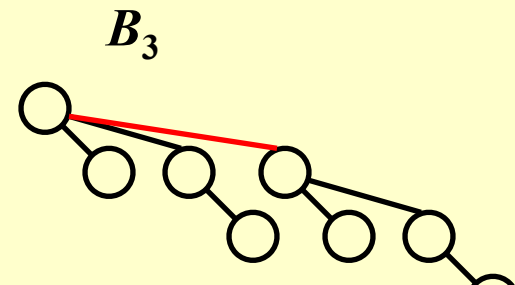
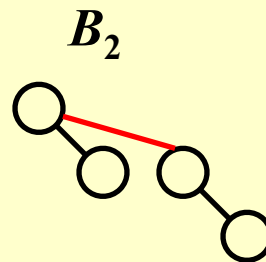
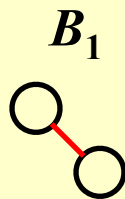
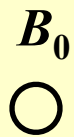
- Review of Binary Heaps
- Leftist Heaps
- Skew Heaps
- **Binomial queues**
- Take-home messages

Structure:

A binomial queue is not **a** heap-ordered tree, but rather a **collection** of heap-ordered trees, known as a **forest**. Each heap-ordered tree is a **binomial tree**.

A binomial tree of height **0** is a one-node tree.

A binomial tree, B_k , of height **k** is formed by attaching a binomial tree, B_{k-1} , to the root of another binomial tree, B_{k-1} .



Observation: B_k consists of a root with k children, which are B_0, B_1, \dots, B_{k-1} . B_k has exactly 2^k nodes. The number of nodes at depth d is $\binom{k}{d}$.

Programming
Techniques

S.L. Graham, R.L. Rivest
Editors

A Data Structure for Manipulating Priority Queues

Jean Vuillemin
Université de Paris-Sud

A data structure is described which can be used for representing a collection of priority queues. The primitive operations are insertion, deletion, union, update, and search for an item of earliest priority.

Key Words and Phrases: data structures, implementation of set operations, priority queues, mergeable heaps, binary trees

CR Categories: 4.34, 5.24, 5.25, 5.32, 8.1



1978 by [Jean Vuillemin](#)

B_k structure + heap order + one binomial tree for each height

➔ A priority queue of **any size** can be **uniquely** represented by a collection of binomial trees.

[Example] Represent a priority queue of size **13** by a collection of binomial trees.

Solution: $13 = 2^0 + 0 \times 2^1 + 2^2 + 2^3 = 1101_2$

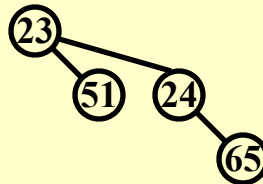
B_0



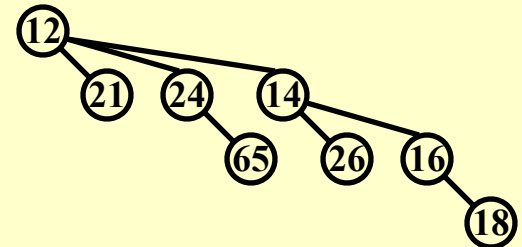
B_1



B_2



B_3

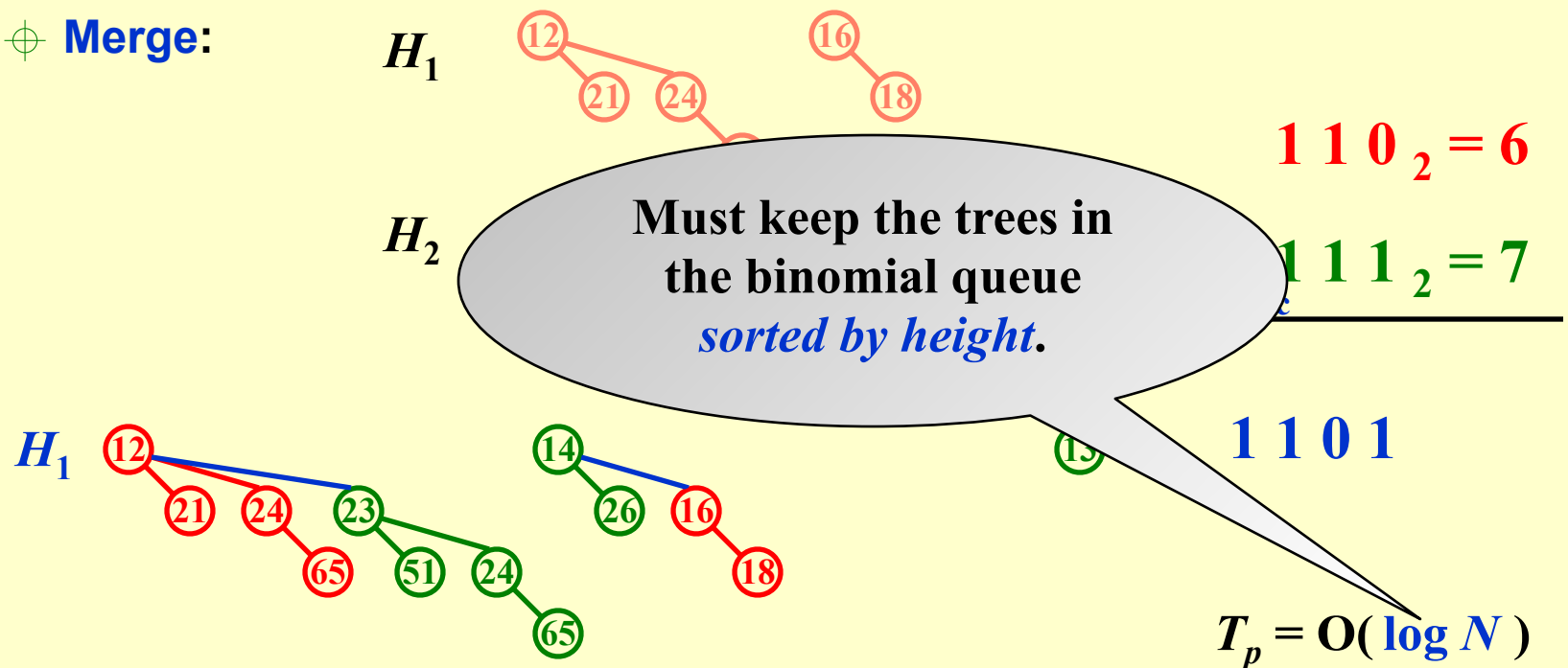


Operations:

- ⊕ **FindMin:** The minimum key is in one of the **roots**.
There are at most $\lceil \log N \rceil$ roots, hence $T_p = O(\log N)$.

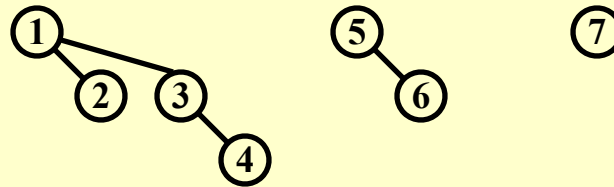
Note: We can remember the minimum and update whenever it is changed. Then this operation will take $O(1)$.

⊕ Merge:



⊕ **Insert**: a special case for merging.

[Example] Insert 1, 2, 3, 4, 5, 6, 7 into an initially empty queue.



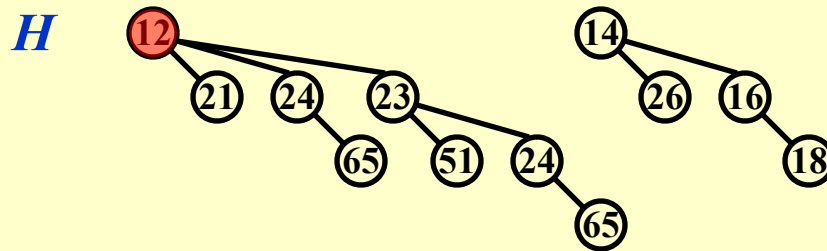
Note:

If the smallest nonexistent binomial tree is B_i , then

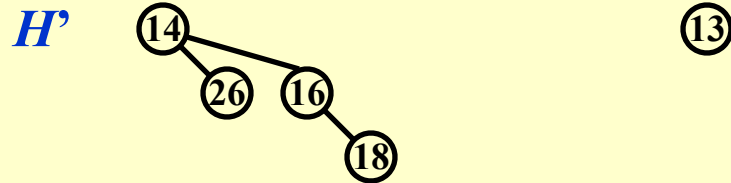
$$T_p = \text{Const} \cdot (i + 1).$$

Performing N **Inserts** on an initially empty binomial queue will take $O(N)$ worst-case time. Hence the **average** time is **constant**.

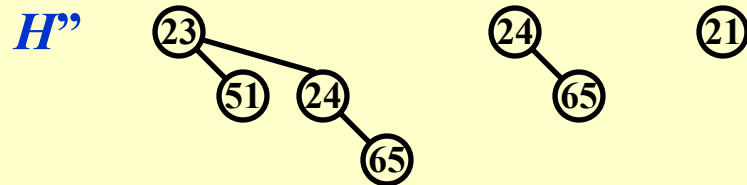
⊕ **DeleteMin** (H):



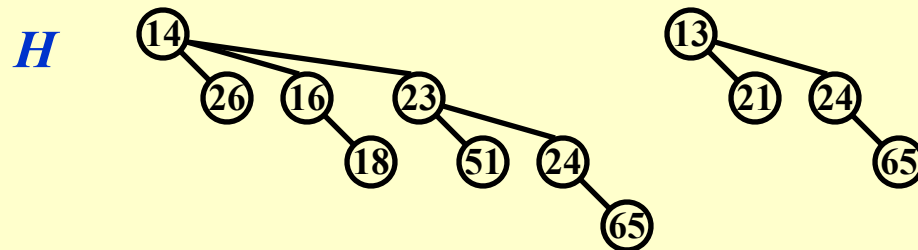
Step 1: **FindMin** in B_k
/* $O(\log N)$ */



Step 2: Remove B_k from H
/* $O(1)$ */



Step 3: Remove root from B_k
/* $O(\log N)$ */

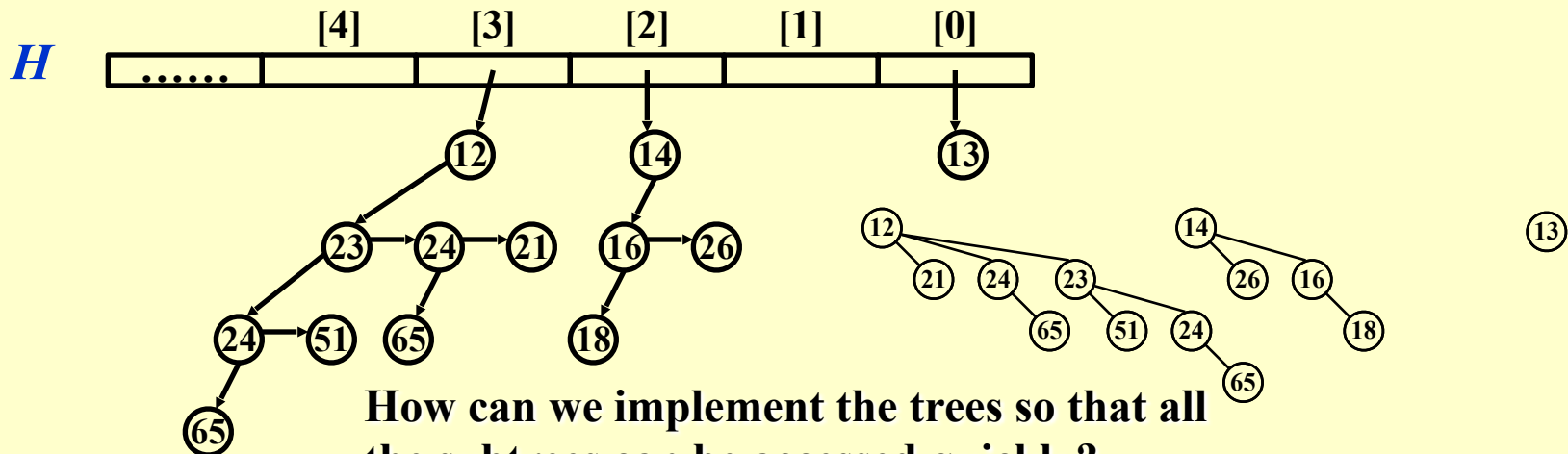


Step 4: Merge (H' , H'')
/* $O(\log N)$ */

Implementation:

Binomial queue = **array** of binomial trees

Operation	Property	Solution
DeleteMin	Find all the subtrees quickly	Left-child-next-sibling with linked lists
Merge	The children are ordered by their sizes	The new tree will be the largest. Hence maintain the subtrees in decreasing sizes



In which order must we link the subtrees?

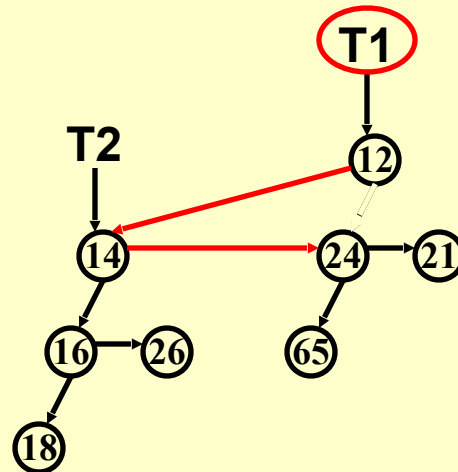
```
typedef struct BinNode *Position;  
typedef struct Collection *BinQueue;  
typedef struct BinNode *BinTree; /* missing from p.176 */
```

```
struct BinNode  
{  
    ElementType    Element;  
    Position       LeftChild;  
    Position       NextSibling;  
};
```

```
struct Collection  
{  
    int            CurrentSize; /* total number of nodes */  
    BinTree        TheTrees[ MaxTrees ];  
};
```

BinTree**CombineTrees(BinTree T1, BinTree T2)****{ /* merge equal-sized T1 and T2 */****if (T1->Element > T2->Element)****/* attach the larger one to the smaller one */****return CombineTrees(T2, T1);****/* insert T2 to the front of the children list of T1 */****T2->NextSibling = T1->LeftChild;****T1->LeftChild = T2;****return T1;****}**

$$T_p = O(1)$$



```

BinQueue Merge( BinQueue H1, BinQueue H2 )
{  BinTree T1, T2, Carry = NULL;
   int i, j;
   if ( H1->CurrentSize + H2-> CurrentSize > Capacity ) ErrorMessage();
   H1->CurrentSize += H2-> CurrentSize;
   for ( i=0, j=1; j<= H1->CurrentSize; i++, j*=2 ) {
       T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i]; /*current trees */
       switch( 4*!!Carry + 2*!!T2 + !!T1 ) {
           case 0: /* 000 */ /* assign each digit to a tree */
           case 1: /* 001 */ break;
           case 2: /* 010 */ H1->TheTrees[i] = T2; H2->TheTrees[i] = NULL; break;
           case 4: /* 100 */ H1->TheTrees[i] = Carry; Carry = NULL; break;
           case 3: /* 011 */ Carry = CombineTrees( T1, T2 );
                   H1->TheTrees[i] = H2->TheTrees[i] = NULL; break;
           case 5: /* 101 */ Carry = CombineTrees( T1, Carry );
                   H1->TheTrees[i] = NULL; break;
           case 6: /* 110 */ Carry = CombineTrees( T2, Carry );
                   H2->TheTrees[i] = NULL; break;
           case 7: /* 111 */ H1->TheTrees[i] = Carry;
                   Carry = CombineTrees( T1, T2 );
                   H2->TheTrees[i] = NULL; break;
       } /* end switch */
   } /* end for-loop */
   return H1;
}

```

```

ElementType DeleteMin( BinQueue H )
{
    BinQueue DeletedQueue;
    Position DeletedTree, OldRoot;
    ElementType MinItem = Infinity; /* the minimum item to be returned */
    int i, j, MinTree; /* MinTree is the index of the tree with the minimum item */

    if ( IsEmpty( H ) ) { PrintErrorMessage(); return -Infinity; }

    for ( i = 0; i < MaxTrees; i++ ) { /* Step 1: find the minimum item */
        if ( H->TheTrees[i] && H->TheTrees[i]->Element < MinItem ) {
            MinItem = H->TheTrees[i]->Element; MinTree = i; } /* end if */
    } /* end for-i-loop */
    DeletedTree = H->TheTrees[MinTree];
    H->TheTrees[MinTree] = NULL;
    OldRoot = DeletedTree; /* Step 2: delete the tree */
    DeletedTree = DeletedTree->LeftChild;
    DeletedQueue = Initialize(); /* Step 3.2: create H' */
    DeletedQueue->CurrentSize = ( 1 << MinTree ) - 1; /* 2MinTree - 1 */
    for ( j = MinTree - 1; j >= 0; j -- ) {
        DeletedQueue->TheTrees[j] = DeletedTree;
        DeletedTree = DeletedTree->NextSibling;
        DeletedQueue->TheTrees[j]->NextSibling = NULL;
    } /* end for-j-loop */
    H->CurrentSize -= DeletedQueue->CurrentSize + 1;
    H = Merge( H, DeletedQueue ); /* Step 4: merge H' and H'' */
    return MinItem;
}

```

This can be replaced by
the actual number of roots

H' */

[Claim] A binomial queue of N elements can be built by N successive insertions in $O(N)$ time.

Proof 1 (Aggregate):

+1	B_0	<u>$/*step = 1 */$</u>
+0	B_1	<u>$/*step = 1, link = 1 */$</u>
+1	$B_1 \quad B_0$	<u>$/*step = 1 */$</u>
-1	B_2	<u>$/*step = 1, link = 2 */$</u>
+1	$B_2 \quad B_0$	<u>$/*step = 1 */$</u>
	$B_2 \quad B_1$	$/*step = 1, link = 1 */$
	$B_2 \quad B_1 \quad B_0$	$/*step = 1 */$
-2	B_3	<u>$/*step = 1, link = 3 */$</u>
	$B_3 \quad B_0$	$/*step = 1 */$
	...	

Total steps = N

Total links =

$$N\left(\frac{1}{4} + 2 \times \frac{1}{8} + 3 \times \frac{1}{16} + \dots\right) = O(N)$$

Expensive insertions **remove** trees, while **cheap** ones **create** trees.

Proof 2: An insertion that costs c units results in a net increase of $2 - c$ trees in the forest.

$C_i ::=$ cost of the i th insertion

$\Phi_i ::=$ number of trees *after* the i th insertion ($\Phi_0 = 0$)

$$C_i + (\Phi_i - \Phi_{i-1}) = 2 \quad \text{for all } i = 1, 2, \dots, N$$

Add all these equations up $\longrightarrow \sum_{i=1}^N C_i + \Phi_N - \Phi_0 = 2N$

$$\sum_{i=1}^N C_i = 2N - \Phi_N \leq 2N = O(N)$$

$$T_{\text{worst}} = O(\log N), \text{ but } T_{\text{amortized}} = 2 \quad \blacksquare$$

See the binary counter problem in CLRS Chap. 16.

Priority queues performance cost summary

operation	linked list	binary heap	binomial heap
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$
ISEMPTY	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$ $O(1)$ †
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$

† amortized

Hopeless challenge. $O(1)$ INSERT, DECREASE-KEY and EXTRACT-MIN. Why?

Challenge. $O(1)$ INSERT and DECREASE-KEY, $O(\log n)$ EXTRACT-MIN.

Priority queues performance cost summary

operation	linked list	binary heap	binomial heap	Fibonacci heap †
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$ $O(1)$ †	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

† amortized

Ahead. $O(1)$ INSERT and DECREASE-KEY, $O(\log n)$ EXTRACT-MIN.

Priority Queues

- Review of Binary Heaps
- Leftist Heaps
- Skew Heaps
- Binomial queues
- **Take-home messages**

Take-Home Messages

- Leftist heaps:
 - Reduce merge cost to $O(\log N)$ by building unbalanced heaps, and put all the computation on the right (light) paths.
- Skew heaps:

Better to implement skew heap with iteration other than recursion since the right path can be of order $O(N)$

 - Avoiding skewness checking by always flipping left and right. Guarantee amortized cost $O(\log N)$.
- Binomial queues:
 - Improve the amortized cost of insertion into $O(1)$. Using the idea of binary counter addition.

Thanks for your attention!
Discussions?

Reference

Data Structure and Algorithm Analysis in C (2nd Edition): Sec. 6.5-6.7, 11.3.

<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/BinomialHeaps.pdf>

<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/FibonacciHeaps.pdf>

<https://web.stanford.edu/class/cs166/lectures/06/Slides06.pdf>

<https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/heaps.pdf>